Malcolm Scott

Bi-directional Quality of Service for an IP tunnel

Computer Science Tripos Part II

Trinity Hall

May 18, 2006

Proforma

Name:	**Malcolm Scott**
College:	**Trinity Hall**
Project Title:	**Bi-directional Quality of Service for an IP tunnel**
Examination:	**Computer Science Tripos Part II, July 2006**
Word Count:	**11236**
Project Originator:	M. Scott
Supervisor:	Dr T. Griffin

Original Aims of the Project

I aimed to implement a system which would allow quality of service policies to be implemented on a relatively slow Internet link such as DSL, using an IP tunnel spanning the link to allow traffic in both directions to be directly controlled. Specifically, existing systems are unable to efficiently schedule simultaneous UDP voice-over-IP and bulk TCP whilst maintaining low latency and packet loss; I intended to implement a system capable of this.

Work Completed

A general purpose bi-directional quality of service system based on an IP-over-UDP tunnel has been implemented. It satisfies the requirements, and maintains demonstrably improved latency and reduced jitter over existing systems, particularly when scheduling concurrent UDP and TCP streams.

Special Difficulties

None.

Declaration

I, Malcolm Scott of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed 

Date May 18, 2006

Contents

List of Figures

Glossary

ADSL Asymmetric Digital Subscriber Line. *See DSL.*

API Application Programming Interface. An interface (a defined set of function calls) which an application can use to interface with a library or system service.

ATM Asynchronous Transfer Mode. A packet switching system designed for use on high speed links, at both data link and network layers, and which provides multiplexed voice, video and data services. ATM works using Virtual Circuits (VCs) to set up a channel for communication between endpoints with guaranteed capacity and delay; ATM therefore has basic QoS functionality built in.

CBQ Class Based Queueing (scheduling algorithm). See Section 2.3.3 (p 9).

CIDR Classless Inter-Domain Routing. A means of dividing an IP address into arbitrary network address and host address portions. An example address in CIDR notation would be 131.111.228.52/24; the "/24" indicates that the first 24 bits (131.111.228) are the network address, and the remainder is the host address within the context of that network.

DRR Deficit Round Robin (scheduling algorithm). See Section 2.3.2 (p 8).

DSL Digital Subscriber Line. A method for providing data services over existing voice telephone lines. DSL uses the unused high frequency capacity of the phone line, allowing analogue telephones to continue to use the low frequency parts unhindered. DSL is subdivided into ADSL (Asymmetric DSL), which has a higher downstream capacity than upstream, and SDSL (Symmetric DSL), which has equal capacity in both directions. The more common type is ADSL, which in the UK is typically available with a downstream capacity of between 0.5 and 8 Mbit/s, and an upstream capacity of up to 0.5 Mbit/s.

FCFS First Come, First Served (scheduling algorithm). Packets are processed and transmitted in the order in which they arrive.

FIFO First In, First Out (buffer). Data is read from the buffer in the order in which it was written.

FQ Fair Queueing (scheduling algorithm). See Section 2.3.2 (p 8).

FTP File Transfer Protocol. A method for transferring files across the Internet. FTP is layered on top of TCP.

GPRS General Packet Radio Service. A method for sending data (in packets) over a radio link, typically a mobile phone. Data rates are typically around 28 kbit/s but depend on factors such as signal-to-noise ratio; latency is typically much higher than DSL and is variable.

HTB Hierarchical Token Bucket (scheduling algorithm). See Section 2.3.3 (p 9).

HTTP HyperText Transfer Protocol. A simple method for transferring files across the Internet. HTTP was originally designed for web browsing but can be used for general-purpose file transfer.

IANA Internet Assigned Numbers Authority. The organisation responsible for allocating IP parameters (addresses, port numbers, protocol numbers, etc.). `http://www.iana.org/`.

ICMP Internet Control Message Protocol. The in-band control protocol for IP, which is mainly used for error reporting and diagnostics.

IP Internet Protocol [1]. See Section 2.1 (p 3).

ISP Internet Service Provider. A company which provides Internet connectivity to customers.

Leaky Bucket A simple burst-smoothing scheduling algorithm. Packets can enter a fixed-size bucket (a FIFO buffer) at any rate, but are constrained to leave (through the "leak") at a fixed rate determined by the link capacity. This is very widely used, partly because it is very simple to implement.

MGCP Media Gateway Control Protocol [2]. A UDP-based VoIP protocol, used by consumer services such as BT Broadband Voice [3].

MTU Maximum Transmission Unit. The maximum size of a frame sent over a network. This will be specific to the network in use, but is typically 1500 bytes on Ethernet. IPv6 specificies a minimum acceptable MTU for any network of 1280 bytes.

OSI Open Systems Interconnect Reference Model. Describes the different layers involved in network communications and the protocols for each layer on the Internet. [4]

QoS Quality of Service. See Section 2.2 (p 4).

PPPoA Point to Point Protocol over ATM. A tunnelling protocol for sending IP packets over an ATM network, for example between a customer and an ISP over a DSL link.

PTB Prioritised Token Bucket. The scheduling algorithm I have devised for this project. See Section 3.3 (p 19).

RED Random Early Detection (queue management algorithm). See Section 2.3.1 (p 8).

RFC Request For Comments. A series of technical documents about the Internet, describing (amongst other things) standards for various protocols. `http://www.rfc-editor.org/`

RR Round Robin (scheduling algorithm). See Section 2.3.2 (p 8).

SFQ Stochastic Fair Queueing (scheduling algorithm). See Section 2.3.2 (p 8).

TBF Token Bucket Filter (scheduling algorithm). See Section 2.3.1 (p 7).

TCP Transmission Control Protocol. See Section 2.1 (p 3).

TOS Type Of Service field [5] in the IP [1] header. See Section 2.2 (p 5).

TUN/TAP Universal TUN/TAP Driver. See Section 2.5.1 (p 11).

UDP User Datagram Protocol. See Section 2.1 (p 3).

VC Virtual Circuit. *See ATM.*

VoIP Voice over IP. Any means of performing a voice conversation over the Internet.

VPN Virtual Private Network. See Section 2.5 (p 11).

WRR Weighted Round Robin (scheduling algorithm). See Section 2.3.3 (p 9).

CHAPTER 1

Introduction

My project concerns the creation of an IP quality of service (QoS) system, which I have called Qtun, to prioritise and control Internet traffic over slow links such as DSL. The scenario I specifically aimed to improve was the simultaneous use of a UDP-based Voice-over-IP (VoIP) system and bulk TCP data transfers such as HTTP or FTP, by minimising packet loss and unnecessary latency on the UDP stream and prioritising it over TCP. I have implemented this using an overlay network based on a custom IP-over-UDP tunnel, and demonstrated that it operates with the desired effect over a standard sub-megabit ADSL connection. My implementation meets, and in some areas exceeds, the core goals outlined in my project proposal.

QoS guarantees are almost entirely absent from current widely-used Internet protocols; TCP will by design aggressively make use of as much of a link's capacity as it can, to the extent that protocols without flow control, such as UDP, will suffer increased latency and dropped packets. Furthermore, the problem is compounded by the large First In, First Out (FIFO) transmission buffers found on almost all DSL modems. These are fitted because they have a beneficial effect on the throughput of bursty traffic. However a side-effect is that when the DSL link is heavily used, the buffers will be full, and therefore packets ready for transmission must wait in the buffer for some time before actually reaching the DSL link. This adds considerable latency — a second or more is common — and makes interactive applications such as VoIP barely useable, especially when combined with packet loss arising from the frequent overflowing of buffers. This effect is even more noticeable on a GPRS connection over a mobile phone; because of the much slower speed, even a small buffer can take a considerable time to be emptied, and I have often noticed latencies of over ten seconds on a heavily-used GPRS connection.

It is possible to avoid this problem by using a simple traffic rate-limiting

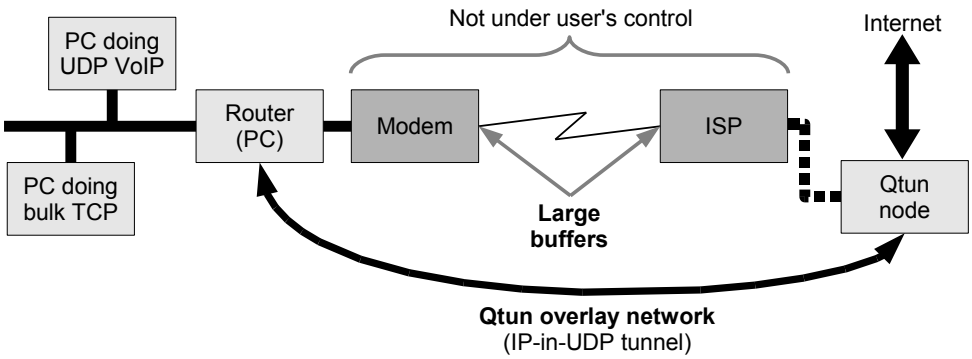

Figure 1.1: Network diagram showing the scenario and my solution

system to ensure that the rate of data entering the DSL modem from a local computer does not exceed the DSL link's maximum transmission rate, hence keeping the modem's buffer almost empty. However there are transmission buffers at both ends of the link, and it is only possible for an end-user to install such a system directly at one end; the other end is under the control of the Internet Service Provider (ISP). Control over the incoming packet stream is needed; existing techniques cannot achieve this effectively without cooperation with the ISP. My assumption is that the ISP will not be cooperative in this regard (typically any QoS policies installed on the ISP's routers will be in their own interests rather than their customers'). In order to implement effective QoS independently of the ISP a purpose-built overlay network with QoS is required; Qtun provides such facilities.

CHAPTER 2

Preparation

This chapter primarily contains a selection of background material on topics relevant to my project.

2.1 Internet protocols

The Internet Protocol (IP) [1], of which the current widely-used version is IPv4 (though IPv6 is scheduled to replace this eventually) is a means of sending datagrams (blocks of data, typically variable-sized up to 1.5 kB) from a source to a destination, optionally via one or more routers if the source and destination are not on the same local network. The datagrams are sent as packets, which prefix a small header to each datagram containing metadata such as source, destination, length and checksum. IP networks make routing decisions on a hop-by-hop basis; each node just decides upon the node the packet should next be sent to. Routes taken by packets from one node to another can therefore change unpredictably at any time.

IP is said to be a data link layer protocol in the Open Systems Interconnect (OSI) Reference Model [4]. IP is not designed to be used directly for applications, as it does not provide a means of multiplexing simultaneous streams of data between two hosts other than by protocol; instead, other network layer protocols providing various different services are encapsulated within the IP packet.

The most common of these protocols are Transmission Control Protocol (TCP) [6] and User Datagram Protocol (UDP) [7]. Both use source and destination port numbers to multiplex simultaneous streams and identify services. UDP just provides a means of sending datagrams; it provides no guarantee that the datagram will arrive in order or even arrive at all. TCP

provides a reliable stream service which, although implemented using unreliable IP packets, is guaranteed and flow-controlled. Any lost packets are retransmitted automatically, duplicate packets are discarded, and packets are reassembled into a stream by the receiver in the correct order. The capacity of the underlying data link is determined automatically and continuously by the protocol in order to alleviate congestion, by gradually increasing the transmission rate until packets are dropped. TCP streams start with a three-way handshake indicating the start of a connection; the connection remains until explicitly terminated.

Whilst it would appear at first that TCP is the superior protocol, UDP is more suited to certain applications. In Voice over IP (VoIP) and other real-time, latency-sensitive systems, if a packet is lost then it is more useful to continue without it (resulting in a brief degradation of quality) rather than introduce latency by waiting for it to be retransmitted. In addition, encapsulating a TCP stream within another TCP stream behaves very poorly and exhibits instability, so when implementing tunnelling (see Section 2.5) it is usual to use UDP or another non-flow-controlled protocol.

The VoIP system I will use for testing (a Fujitsu FDX-840 hardware telephone-to-IP adapter with a BT Broadband Voice [3] connection) uses MGCP, the Media Gateway Control Protocol [2], layered over UDP. This is a simple protocol with an open standard which is easy to inspect programmatically. However, my intention is to make Qtun sufficiently general-purpose that it will work with any VoIP system.

2.2 Quality of Service

Quality of service (QoS) is a generic term referring to any guarantee which can be provided by a network layer to higher-level services. Such guarantees are typically stated in terms of latency or throughput, and could be expressed as anything from a strict set of bounds ("the available throughput will exceed 1 Mbit/s") to an indication of an order of priority ("interactive traffic will be prioritised over bulk transfers"). The Internet needs QoS because of its multi-purpose nature: many different types of traffic can traverse the Internet and contend for capacity, and some types of traffic are considered by the user to be more urgent than others. For example a VoIP telephone call made across the Internet will suffer considerably from high latency, as this will result in a time lag in the conversation (and it has been shown that a time lag of over 250 ms in a conversation is intolerable to humans [8]); bulk downloads merely take very slightly longer under high latency. Furthermore, there are some types of traffic for which there is no "correct" QoS guarantee: web browsing traffic

may be seen as either interactive or bulk, and this will depend in part upon the individual user's preference. It is desirable for a user to be able to control how traffic is prioritised on his or her own link to the Internet.

Strict bounds require every link and every device in the network to co-operate in deciding upon and enforcing the guarantee, as any part of the network could act as a bottleneck; the network architecture and protocols need to be designed from the outset to accommodate this. ATM is a protocol designed with this in mind; when setting up an ATM virtual circuit (VC) between two endpoints, negotiation of the capacity of that VC takes place and this capacity is enforced for its lifetime. IP, however, has only very limited QoS capabilities. The IP packet header does have a Type of Service (TOS) field [5], intended to provide a vague hint to routers on how individual packets should be treated (for example, that latency should be minimised, or throughput maximised) but few routers actually obey these hints — and because of this, few applications make use of the TOS field.

For this reason, the Internet as a whole can provide no QoS guarantees. Individual routers can be configured to perform QoS techniques according to configured policies, but any guarantees arising will typically apply only to individual links in one direction; end-to-end guarantees are not possible. Even if we just consider communications between two fixed endpoints, due to the unpredictable nature of Internet routing the provision of QoS guarantees in such communications would require cooperation between the owners of every possible router between the two endpoints. Furthermore, packets will likely traverse routers belonging to several ISPs, who may have configured conflicting policies on their routers.

The core of the Internet currently has plenty of spare capacity however, so working without QoS between core routers is acceptable until high-bandwidth services such as streaming video become more widespread. Between an end user and a server, the bottleneck will almost certainly be the user's relatively slow link to his or her Internet service provider (which, for the majority of home users in the UK, will be an ADSL or cable link with a downstream capacity of between 0.5 and 8 Mbit/s and a lower upstream capacity). Tightly controlled prioritisation and packet scheduling is therefore most crucial on this link, at the edge of the Internet.

Some ISPs do apply traffic shaping on their customers' links, but generally this is heavily influenced by their own interests: for example, with the intention of reducing the amount of peer-to-peer file-sharing traffic on their network, and hence avoiding the increased costs associated with maintaining a higher-capacity network to cope with demand. It is more frequent to find users complaining about their ISP's QoS system than praising it, not least

because the end user has no control over the system.

2.2.1 Transmission vs. reception

It is important to note that QoS techniques are much more effective at controlling transmitted data than received data. Most attempts to control data received over a slow link will cause the transmission buffers at the other end of the link to remain full, and will not solve any latency problems. It is therefore necessary to install QoS systems with matched policies at both ends of the link; this is the purpose of my project.

Techniques for controlling some types of incoming traffic do exist. It is possible, for example, to cause the transmission rate of a TCP stream to be reduced by its source simulating congestion which will be detected by the flow control algorithm, i.e. by dropping packets temporarily. Such techniques are referred to as ingress policing algorithms; a simple example would be an algorithm which allows incoming flows to be received up to a certain rate, and drops any packets which arrive faster than this. RED (see Section 2.3.1) can be used as a more sophisticated ingress policing algorithm. However it is important to remember that such schemes are inefficient as some of the link's capacity will be used by packets which are intentionally dropped. It also takes time for the source to detect the packet loss, and therefore ingress policing algorithms are slow to respond to changes.

2.3 Techniques for providing QoS guarantees

There are several broad techniques for providing QoS, which have different effects and are often used in combination; such a combination is sometimes referred to as a traffic shaping setup. A summary of the most common techniques follows.

A flow is defined as any stream of packets corresponding roughly to one conversation between two endpoints. For connection-based protocols such as TCP this is well-defined (an established TCP stream is a single flow), whereas in datagram-based protocols such as UDP the definition is more open to interpretation (most commonly, the flow is identified by the tuple of source and destination addresses, optionally including the source and destination port numbers).

2.3.1 Rate limiting of a single flow

Rate limiting (also known as throttling) is the limitation of the maximum data rate of a packet stream. This is applied to all data passing over a link in order

to limit the total transmission rate of traffic — rate limiting algorithms assume the existence of a single flow (though adaptations exist to handle multiple flows or classes of traffic, which are discussed later). Rate limiters are typically first come, first served (FCFS), i.e. they do not intentionally reorder packets (though of course another part of the QoS system could perform reordering).

A modem can be considered as a throttling device, as it may have to cope with packets arriving from a fast network (which could be, for example, 100 Mbit/s Ethernet) at a higher rate than the output (for example, DSL) is capable of. It achieves this by having a fixed-size FIFO buffer into which incoming packets are placed; packets are transmitted from this buffer as fast as possible, but if the packet arrival rate is not sustainable then the buffer will eventually become full and subsequently received packets will be dropped until buffer space frees up. Such a setup is an example of the Leaky Bucket algorithm (in which the FIFO buffer acts as the "bucket"). This has the effect of smoothing out bursts in traffic, which is achieved well; the larger the buffer, the larger the bursts which can be smoothed. However, when the buffer contains a large amount of data, packets will take a considerable time between entering the buffer and being transmitted; in other words, latency will suffer badly. As previously discussed, high latency causes problems for interactive applications.

It is assumed that the behaviour of the modem cannot be changed by the user, as it would require an extensive redesign of the modem's hardware. So if the latency problem is to be solved, the modem's throttling behaviour must be controlled by some means external to the modem. This is possible by limiting the rate at which data enters the modem from the faster link: if this data is already at a rate sustainable by the slower link, the buffer will only ever contain packets currently arriving or being transmitted; packets will not sit idle in the modem. A throttling device is therefore required between the modem and the fast network; this can have more complex policies in place of the FIFO policy of the modem which can avoid the drawbacks of the leaky bucket algorithm.

A more flexible approach to rate limiting is the Token Bucket Filter (TBF) algorithm (sometimes just referred to as token bucket; not to be confused with leaky bucket). In this algorithm, tokens arrive into a bucket at a fixed rate which determines the desired transmission rate. The transmission of packets requires the expenditure of a number of tokens (typically one token per byte); if insufficient tokens are available the packet is delayed, often by queueing in a FIFO as in the leaky bucket algorithm. The token bucket can only store a fixed number of tokens, beyond which incoming tokens are dis-

carded; the bucket size determines the maximum size of a burst of data which can be transmitted after the link has been idle, before settling into a steady transmission rate. Where low latency is important, the maximum burst size must be small to avoid filling up the modem's buffer.

The aforementioned algorithms only drop packets when a buffer is full. An alternative approach is taken by the Random Early Detection (RED) [9] algorithm, which attempts to maintain short queues (in order to keep latency low) but still allow occasional bursts of packets. The algorithm does not perform rate limiting on its own, but is often combined with a rate limiting algorithm such as leaky bucket or token bucket. RED works by dropping packets early according to a function that depends upon the current queue length. Above a threshold, the probability of dropping a packet increases linearly with the size of the queue, such that a full queue drops a packet with a probability of 1, and an empty queue drops no packets. This is particularly suited to flow-controlled protocols such as TCP, in which the transmission rate is determined based upon which packets are dropped; RED's dropping of packets before the queue is full results in the transmission rate reaching the maximum available capacity more quickly (rather than repeatedly overshooting and causing a large number of packets to be dropped) and remaining stable.

2.3.2 Scheduling of multiple flows

When multiple flows are transmitting simultaneously, it is desirable for them to be able to share the link fairly — such that flows are all allowed the same proportion of the throughput of the link. The scheduling of individual flows is beyond the scope of my project, although I am including a brief overview here as I have adapted techniques originally designed for such scheduling.

The standard policy of FCFS does not provide a fairness guarantee. Although it tends to be approximately fair on average, there exist combinations of circumstances which will cause a bias towards certain flows. Nagle [10] suggested Round Robin (RR) scheduling of flows; this is fair if packets have a constant size, which is rarely the case on IP networks. Fair Queueing (FQ) [11] is an algorithm which achieves near-perfect fairness, but this scales poorly when there are many flows present; it requires $O(\log(n))$ work per packet, where n is the number of flows. Stochastic Fair Queueing (SFQ) [12] simplifies the FQ algorithm such that flows are grouped together into buckets according to a hash function, and so n is smaller than the number of flows (and may be constant), but this compromises on fairness as flows in the same bucket can still interfere with one another.

Deficit Round Robin (DRR) [13] provides similar guarantees to FQ, using

a much cheaper algorithm in which the required work per packet remains constant. DRR is based upon RR, but compensates for RR's unfairness with variable-sized packets by keeping track of the past unfairness ("deficit") the algorithm has caused each flow, and uses this deficit to determine how much data the flow is allowed to transmit on the next cycle. This can be considered a multiple-flow version of token bucket, in which a flow's deficit counter corresponds to the number of tokens available to that flow.

2.3.3 Class-based scheduling

It is often useful to treat different types of traffic in different ways in order to provide different sorts of guarantees to different applications. Such a system will require a packet classifier which inspects packets, typically by looking at fields in the headers at various levels (in the simplest case just the IP and TCP/UDP headers are used, which provide such information as the packet's source, destination, port and length) but more complex classifiers will dissect protocols encapsulated within the TCP stream or UDP datagram. According to configured conditions the packet will be placed into one of several classes; different classes are treated in different ways by the traffic shaper.

It would be possible to configure the system such that each class has an independent rate limiter such as token bucket, with round-robin scheduling between classes such that each class has a chance to transmit one packet in turn. This would effectively allow the specification of maximum rates for each type of traffic. However this setup is likely to waste capacity: if one class is idle, then its capacity remains unused and cannot be transferred to other busy classes. An algorithm is required which can take account of the existence of multiple classes of which a subset are trying to transmit at unspecified rates at any given time.

Class-based Queueing (CBQ) [14] is such an algorithm, which uses a Weighted Round Robin (WRR) scheduler to transmit packets from different classes in order of priority or according to limits on average and burst transmission rate. CBQ can also throttle the combined total bandwidth use of all classes by measuring and controlling the average idle time of the link: for example when a 2 Mbit/s connection is throttled to 1.9 Mbit/s, CBQ ensures that the link is idle 5 % of the time. The drawback of CBQ is that it requires accurate, high-resolution timing, and also exact knowledge of the capacity of the link (which may in reality be unknown or variable). These problems are illustrated in the context of the Linux Traffic Control system (see Section 2.4.1) in the `tc-cbq-details` manual page [15].

Hierarchical Token Bucket (HTB) [16] is a class-based adaptation of token

bucket. It was originally devised as part of the Linux Traffic Control system, although the algorithm has since been used for a wide variety of other applications such as IEEE 802.11 wireless networking [17]. This is easier to implement than CBQ as it does not depend on interface characteristics, yet achieves the same goals as CBQ. HTB works with a hierarchy of classes; incoming tokens are distributed between these classes in accordance with configured priorities and limits. Most importantly, classes can take spare tokens from other classes which are transmitting at less than their permitted rate.

2.4 Existing QoS systems

Several systems already exist which can perform traffic scheduling in order to provide QoS guarantees. Usually this is implemented as part of a larger general-purpose system. Traffic scheduling is almost always done on routers; routers may be implemented in software (for example, the Linux operating system has facilities to become a fully-featured Internet router with QoS capabilities) or in hardware. Software routers are usually only used on a small scale, for example to connect a home network to the Internet, or where capabilities are required which are not found in affordable hardware routers, such as a VPN (see Section 2.5). Several "hardware routers" sold for home use are in fact software routers, and furthermore many of them run Linux. My project has been done in the context of a Linux-based software router, as such routers are easily extended using user-space Linux applications.

2.4.1 Case study: Linux Traffic Control

The Linux kernel has a very flexible QoS architecture called Traffic Control, which is controlled using the user-space utility `tc`. This works by associating a "qdisc" (queueing discipline) with each interface, on which packets to be transmitted are enqueued by the kernel. The default qdisc is `pfifo_fast`, a FIFO implementation. Other simple qdiscs cover most of the algorithms I described above: `sfq`, `tbf`, and `red`. More complex qdiscs are hierarchical and create a tree which may contain other child qdiscs (of any type, including another of the same type) into which packets are classified according to a set of rules; these include `prio` (which has a numeric priority assigned to each child, and passes control to the children in order of priority; this is approximately equivalent to WRR), `cbq` and `htb`. A simple ingress policing system is also implemented, which can drop packets of flows which arrive above a certain rate.

Traffic Control is a very effective system as it stands. However it is not well-suited to prototyping new systems which require additional code as it runs entirely inside the kernel; making changes to the running code requires restarting the entire system.

2.5 Tunnelling and overlay networks

Tunnelling refers to the encapsulation of a data link (or lower) layer protocol, such as IP or Ethernet, inside another data link (or higher) layer protocol, such as IP again or UDP. This is sometimes used to allow data for one network to travel across another, incompatible network. For example, DSL connections are usually natively ATM-based, but the PPPoA encapsulation protocol is used to allow IP traffic to flow over this link between the DSL customer and the Internet. Similarly, IPv6-in-IPv4 tunnels (either explicit or implicit) are used to connect disparate IPv6 networks together over existing IPv4 infrastructure [18, 19].

Another common use of tunnelling is in Virtual Private Networks (VPNs). VPNs typically tunnel IP over IP (or occasionally Ethernet over IP), but with an encryption and authentication layer. This has two major uses: to protect data sent over the VPN from interception, and to allow a computer to appear within another network in an authenticated manner (for example a company employee could connect to a company VPN from a computer at his or her home in order to appear within the company's internal network and hence communicate with other internal systems). There is a very wide variety of VPN software, and even of VPN protocol standards, in existence. Open source Linux VPN applications include Poptop PPTPd [20], OpenVPN [21], tinc [22] and an IPsec [23] implementation in the kernel; each of these uses a different protocol and has very different modes of operation.

A network which is in whole or in part constructed from tunnels layered over other networks is known as an overlay network, as it hides the detail of the underlying network from applications.

2.5.1 APIs for tunnelling

Linux includes a driver — the Universal TUN/TAP Driver [24] — to allow the kernel to offload network functionality to a user-space application. This is accessed by using standard C library calls (`open`, `ioctl`) on a special device file (`/dev/net/tun` or `/dev/tun`) and will create a network interfaces to which other applications can send packets in the same way they would send packets over a physical network. The tunnelling application can use the `write` and

`read` functions (or similar) to send a packet onto the network via the kernel, and receive a packet which another application wishes to send, respectively. A similar driver with the same interface is present in other UNIX variants such as FreeBSD and Solaris.

CHAPTER 3

Implementation

3.1 System design

3.1.1 Operating system and language

I will be implementing Qtun as a user-space Linux application, due to the popularity of Linux in software routers (see Section 2.4). Linux is open source so problems encountered with interacting with the kernel or libraries may be resolved by inspecting its source code. In addition, Linux has the benefit of providing APIs such as the universal TUN/TAP driver (Section 2.5.1), which will make the integration with the kernel networking subsystem considerably easier and more elegant. Other open source UNIX variants such as FreeBSD would also satisfy these criteria; however I have very little experience of these.

The use of Linux and the universal TUN/TAP driver introduces a requirement for my project to be based upon the C programming language. Derivative languages such as C++ which allow direct calls to C library functions would also be possible; however since a considerable portion of the code is to be centred around C API calls I would have little opportunity to use the higher-level features such languages provide. Furthermore use of C has the benefit that the compiled code will be compact, fast, and not dependent on the substantially-sized standard C++ libraries — thus making the code portable to embedded systems such as home ADSL routers which typically have tight constraints on memory and storage.

I have limited experience of programming in C, so my first steps on the project will be to gain sufficient familiarity with the language.

3.1.2 Tunnelling

Upon first researching possibilities for implementation of this project, I decided to implement it as an add-on to an existing VPN system. This quickly proved to be much more complex than I had expected, as the VPN applications I investigated did not lend themselves to easy modification of their queueing behaviour, and furthermore it is possible that such modifications would affect the security of the system in unexpected ways. Instead I came to the conclusion that implementing my own overlay network from scratch would be simpler, and also more flexible as it would not tie the user to a particular VPN. If the security of a VPN is required, it is possible to layer Qtun's overlay network over most other Linux VPNs.

In order to provide the overlay network between Qtun nodes, I will implement a custom IP-in-UDP encapsulation protocol. This will prefix onto the IP packet a short identification header containing a magic number and protocol version number, and place the result in a UDP packet for transmission across the Internet. The magic number and protocol version are checked by the receiver and must both match the expected values; the purpose of this is to avoid erroneous packets being output when incompatible versions of Qtun are attempting to communicate. The protocol is illustrated in Figure 3.1. A custom protocol will be used because there is no commonly-implemented standard protocol which would be suitable; in any case the tunnel will only ever exist between two Qtun nodes and no interoperability with other tunnelling systems is required.

The universal TUN/TAP driver provides a means of creating a standard network interface and transferring packets to and from the kernel. Once a packet is received from the kernel it will be passed to the queueing and QoS module; packets output from that module then become encapsulated within a UDP packet and sent onwards to the remote node. The remote node will strip off the UDP header and pass the packet directly onwards to the TUN/TAP driver — all QoS is performed by the transmitting node for the reasons discussed in Section 2.2.1. The flow of a packet through the system

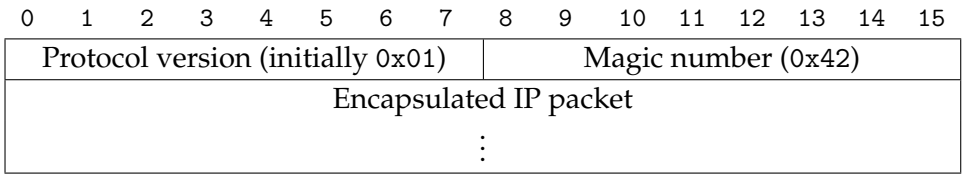

Figure 3.1: Encapsulation packet structure

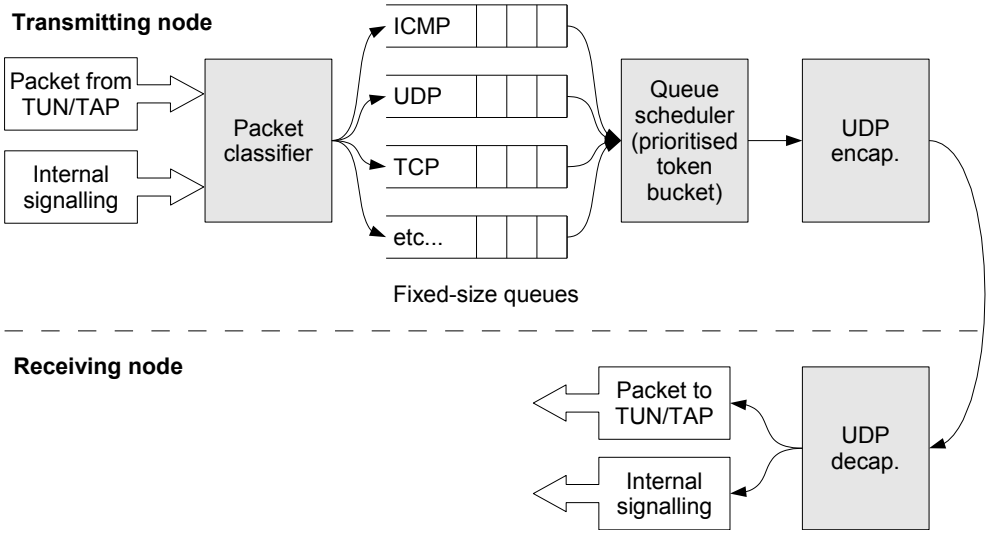

Figure 3.2: Packet flow through the system

is illustrated in Figure 3.2.

The code for the encapsulation module will be in `encap.c`, and the TUN/TAP driver interface will be `tun.c`. Together these modules will manage the Qtun overlay network.

3.1.3 Queueing and QoS module

The queueing and QoS module — `queue.c` — will be responsible for enqueueing packets to transmit, and scheduling the dequeueing of such packets according to a scheduling algorithm. This module will also contain a packet classifier in order to select the correct one of a number of queues into which a given packet should be placed; different queues will be treated in different ways by the scheduler. As discussed in the previous chapters it is important to keep queue sizes small in order to minimise latency, and to ensure that the scheduler does not transmit data at a higher rate than the bottleneck link can sustain.

This module must be implemented in a general-purpose manner, in order that a wide range of QoS policies could be implemented. This is important as the desired policy will vary greatly between different users and networks. Although Qtun will be implemented with a particular scenario in mind — the sharing of an ADSL link between UDP VoIP and bulk TCP streams — it will not be restricted to such a scenario and aims to be easily modifiable to solve a wide range of different problems.

The packet classifier should be able to discriminate packets based upon various criteria of the packet, such as source or destination address, transport layer protocol (such as TCP or UDP), source or destination port number, packet length (according to lower and upper bounds). These can all be obtained from a packet by looking at fields at fixed locations in the IP, TCP or UDP packet header. In addition, it may be useful to provide a facility to match arbitrary strings at fixed locations within the packet data, in order to inspect higher-layer protocols which are not implemented in the classifier directly; for example, this could be used to inspect fields in MGCP packets to distinguish incoming from outgoing VoIP calls.

Various approaches for implementing queue scheduling algorithms were discussed in Section 2.3. Of these the most suitable would be a class-based variant of token bucket, such as HTB, as this allows accurate control of both the total transmission rate and the priorities of traffic within each class, without requiring high resolution timing (as this is not available to a user-space application on a preemptive operating system such as Linux).

Rather than using HTB itself, I have opted for a simpler variant which would be easier for the user to understand and is slightly less computationally intensive; the full capabilities of the HTB algorithm are almost always unneeded. HTB operates upon a hierarchy of classes. My algorithm — which I shall call Prioritised Token Bucket (PTB) — instead maintains a linear linked list of classes, which are allocated tokens in order of priority. I will discuss the PTB algorithm in more detail in Section 3.3.

3.1.4 Internal signalling

In some situations it may be necessary for the two endpoints of the overlay network to communicate events to each other. This will be done in-band such that the signalling packets pass through the same scheduling algorithm as data packets, in order that the total transmission rate can be controlled (though it may be useful to prioritise signalling packets such that they are always transmitted in favour of data). This will be handled by `rmi.c`, which will implement a custom remote method invocation (RMI) protocol. RMI packets will be marked as protocol 114 in the IP header (this protocol number is reserved by the IANA for "any zero-hop protocol"; RMI packets will not leave the Qtun overlay network so satisfy this criterion). Packets will be passed to `rmi.c` by `encap.c` upon decapsulation of a packet of that protocol number. Figure 3.3 shows the detailed structure of an RMI packet.

The protocol will not provide any guarantees about message delivery directly; in this way it is similar to IP or UDP. For this purpose, I will im-

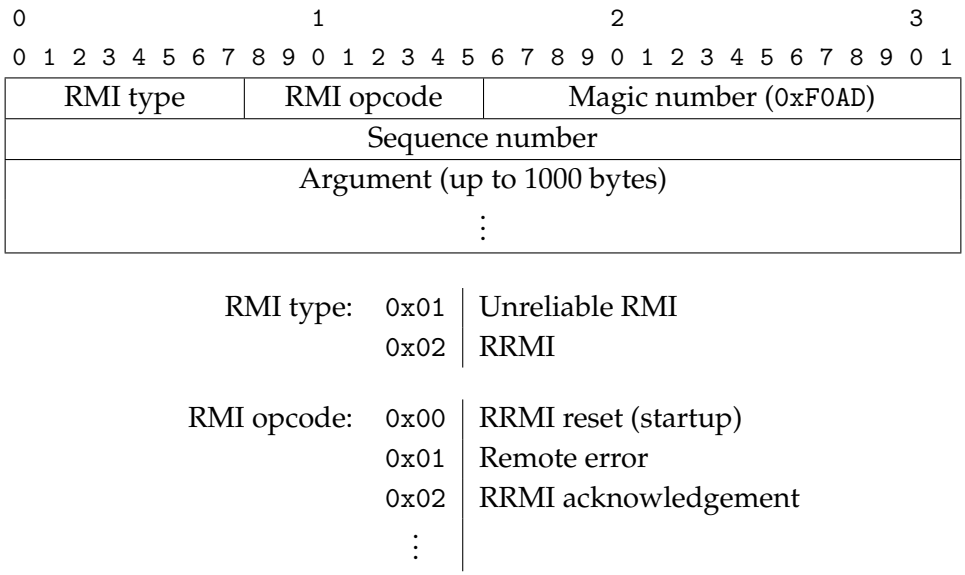

Figure 3.3: RMI request structure

plement a Reliable RMI protocol (RRMI) on top of RMI, which will handle acknowledgement of requests, retransmission of lost requests, etc..

3.1.5 Input/output abstraction layer

In order to avoid duplicated code for performing input and output operations on file descriptors, all such operations will be handled by an abstraction layer, `io.c`. This will allow the other modules, such as the TUN/TAP driver interface, to register file descriptors (FDs) for reading and/or writing (by calling the `io_add_readfd` or `io_add_writefd` function respectively), and will handle the mechanics of reading or writing data from or to the relevant FD when needed. This module is described in more detail in Section 3.4. The interaction of the core modules is illustrated in Figure 3.4.

3.1.6 Other peripheral modules

There will be, in addition, modules to handle peripheral tasks such as the logging of messages for debugging or for informing the user about error conditions (`log.c`), and startup and initialisation of the other modules (`main.c`).

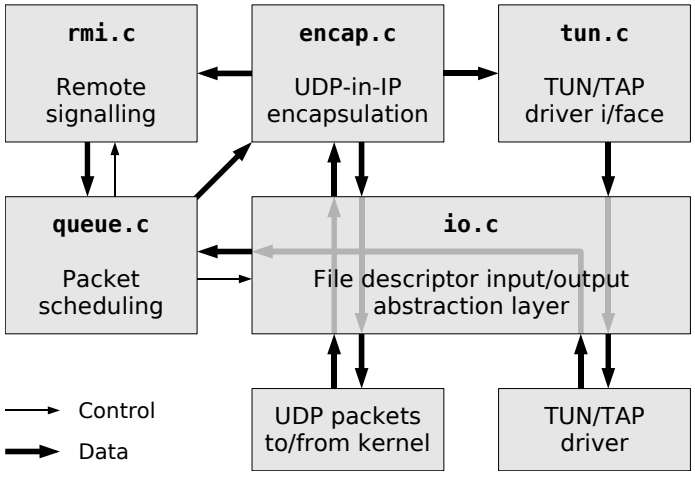

Figure 3.4: Interaction of the core modules

3.2 Implementation approach

3.2.1 Toolchain and libraries

In order to allow Qtun to be ported to any hardware platform on which Linux will run, I will be using the standard GNU C compiler, `gcc` [25]. I will not target any particular version of the compiler, as I will myself be testing Qtun on several different computers, which may have different versions of `gcc` installed. The build process will be managed by SCons [26]. I chose this over the more usual `make` because of its simplicity: with SCons it is possible to just list the source files which should be compiled, and the correct build process will be determined automatically (though if required, SCons can provide more detailed control; it is configured using short Python scripts and is hence very flexible).

The only libraries I will need to use will be the GNU C libraries [27]. These are found on almost all Linux systems. The Linux kernel header files will be required for building Qtun, due to the use of the universal TUN/TAP driver whose interface is defined in these headers.

3.2.2 Version control

I will use Subversion [28] for version control. This has several benefits over the more commonly-used CVS, the most useful to me being support for changesets — i.e., changes to multiple files are committed together atomically, which makes keeping track of related changes to different modules

easier. Trac [29] will be used as a front-end to the Subversion repository and to maintain a list of defects as I find them during testing.

3.2.3 Testing

As far as is possible, each module will be tested individually as it is completed. To do this I will make use of existing completed modules and skeleton implementations of the modules which are still to write.

My approach to testing the complete finished system will involve both subjective and objective testing. For subjective testing, a setup with a VoIP telephone on an ADSL line with simultaneous TCP transfers can be tried by users both without and with Qtun; the users would judge whether the audio quality and perceived time lag on the conversation is acceptable.

The bulk of my testing, however, will be objective and quantitative. This will involve the use of utilities to simulate various types of flow which will pass through the system, and to gather statistics on the round trip time and packet loss. As the ADSL line I have access to for testing does not have VoIP equipment installed, I will implement a simulator which generates a UDP traffic flow similar to that produced by a common VoIP system, and reports statistics. I will produce graphs showing the throughput of different flows passing along the line, both with and without a Qtun setup, in order to illustrate adaption to changing situations.

The testing procedures will be detailed in Chapter 4.

3.3 Prioritised Token Bucket algorithm

The PTB algorithm forms the core of Qtun's QoS system. This is a class-based token bucket filter (TBF) algorithm I devised for Qtun which draws some ideas from hierarchical token bucket, although the implementation is very different. PTB maintains a linked list of packet queues (`queue_list`) in priority order. Queues have the structured data type `queue_node`, which contains the following fields:

`guaranteed_rate`**:** the data rate (in bytes per second) which the algorithm will dedicate to this queue; lower priority queues are constrained to use the remainder when this capacity is subtracted from the total available capacity of the link. If this queue does not make use of all of this reservation, some capacity may be wasted. This could be used to ensure that, for example, a small portion of the link is always reserved for low rate interactive traffic.

`ceiling_rate`: the maximum data rate at which this queue can transmit; above this rate, lower-priority queues are allowed to transmit instead. This is useful in order to limit the use of high-priority queues, and avoid them starving lower-priority queues of capacity.

`max_queue_size`: the maximum amount of data this queue can contain, in bytes. This must be small in order to keep latency to a minimum.

`excess_mode`: the behaviour of packets arriving for the queue which would cause it to become larger than `max_queue_size`. My implementation has `DROP` as the only option here — such packets are discarded — but the field is present in order to allow future extensions. A potential alternative option could, for example, close connections using an ICMP error code (though this would be of very limited use).

`queue_head`, `queue_tail`: pointers to both ends of a linked list of packets in the queue. Packets are added at `queue_tail` and transmitted from `queue_head`.

`queue_size`: the current size of the packet queue, in bytes.

`guarantee`: used for temporary storage by the algorithm; contains the number of tokens guaranteed to this queue in the current quantum (based on `guaranteed_rate`).

`permit_tokens`: temporary storage; contains the number of tokens the queue is permitted to use in this quantum (based on `ceiling_rate`).

`permit_exceeded`: temporary storage; a flag which when set indicates that this queue cannot send another packet during this quantum as that would cause the `ceiling_rate` to be exceeded.

`name`: a textual identifier for the queue, for printing in debugging messages.

In addition, a global counter (`sparetokens`) is used to store the current number of tokens available for use.

The algorithm itself is described in detail in Figure 3.5. It works by dividing time into quanta (the length of a quantum is customisable by setting the `QUANTUM` constant, but I have found that 10 milliseconds works well at DSL speeds). At the start of each quantum, the spare token counter is incremented by a fixed number (`TOKENRATE`). One token can be used to transmit one byte, i.e. a packet n bytes long requires n tokens to be expended in order for the packet to be transmitted; packets must be transmitted whole. Some

```
1  sparetokens = 0;
2  repeat at the start of each quantum
3      sparetokens += TOKENRATE;
4      foreach queue do
5          if queue.guaranteed_rate > 0 then
6              queue.guarantee += queue.guaranteed_rate × QUANTUM;
7              if queue.guarantee > GUARANTEEMAX then
8                  queue.guarantee = GUARANTEEMAX;
9              sparetokens −= queue.guaranteed_rate × QUANTUM;
10         else queue.guarantee = 0;
11         if queue.ceiling_rate > 0 then
12             queue.permit_tokens += queue.ceiling_rate × QUANTUM;
13             if queue.permit_tokens > PERMITMAX then
14                 queue.permit_tokens = PERMITMAX;
15         else queue.permit_tokens = 0;
16         queue.permit_exceeded = false;
17     repeat
18         queue = first non-empty queue without permit_exceeded set;
19         if queue exists and head packet length ≤ sparetokens + guarantee then
20             if queue.ceiling_rate > 0 and packet length ≥ queue.permit_tokens then
21                 queue.permit_exceeded = true;
22             else
23                 if queue.guarantee > 0 then
24                     queue.guarantee −= packet length;
25                     if queue.guarantee < 0 then
26                         sparetokens += queue.guarantee;
27                         queue.guarantee = 0;
28                 else sparetokens −= packet length;
29                 if queue.ceiling_rate > 0 then
30                     queue.permit_tokens −= packet length;
31                 send packet to encapsulation module;
32     while previous iteration sent at least one packet ;
33     if sparetokens > TOKENMAX then sparetokens = TOKENMAX;
```

Figure 3.5: Pseudocode for the PTB algorithm

of these tokens are preallocated to queues with guaranteed rates (lines 5–10 of the pseudocode), and ceiling rates are converted into a number of tokens each queue is permitted to use (lines 11–15).

After the preallocation of tokens, the algorithm finds the first non-empty queue which has not exceeded its ceiling rate. The packet at the head of this queue is transmitted if and only if there are sufficient tokens either in the global spare token counter or in the per-queue preallocated token counter to transmit the packet, and if transmitting the packet would not cause the queue's ceiling rate to be exceeded (lines 19–31). Multiple packets can be transmitted per quantum, by repeating this Section of the algorithm until no further packets can be transmitted.

The algorithm permits a degree of burstiness by allowing a limited number of unused tokens to be carried over from previous quanta. This applies to both the global spare token counter, and to the preallocated per-queue token counters. The points at which these counters will saturate are controlled individually, using the constants `TOKENMAX` (for `sparetokens`), `GUARANTEEMAX` (for `guarantee`) and `PERMITMAX` (for `permit_tokens`). I have set these to be small in order to err on the side of low latency rather than burstiness, but the algorithm provides the facility to allow more burstiness for other scenarios if necessary.

3.3.1 PTB implementation details

In Qtun, the PTB algorithm forms the main loop, which runs continuously during normal operation. As such, it calls various other modules' housekeeping functions periodically — in particular, `rrmi_check` which performs retransmission of unacknowledged RRMI requests if necessary, and `io_select` which processes any pending transfers and callbacks in the I/O abstraction layer. The latter can be used to pause program execution until either data arrives on a file descriptor or a timer elapses; this is therefore used to wait at the end of each quantum.

A packet enters `queue.c` through the `enqueue_packet` function. This calls the packet classification function, `classify_packet`, which iterates through a linked list of packet filters (of structured data type `classifier_entry`) comparing the packet against the criteria in each. The first classifier entry which the packet fully matches will be used to determine the queue to which the packet will be assigned (or alternatively, will provide a function which will be called on match). Each classifier entry contains the following fields:

`src_ip`, `src_netmask`, `dst_ip`, `dst_netmask`**:** criteria for matching the source or destination IP address. The relevant IP address from the packet's

header is first masked using the `src_netmask` or `dst_netmask` (using a bitwise AND) before being matched against `src_ip` or `dst_ip`. This allows the matching of IP addresses within a given network, as in the CIDR addressing scheme, rather than having to match the IP address exactly. A zero value of the netmask is a wildcard and indicates that any IP address will match.

`src_port`, `dst_port`: criteria for matching the TCP or UDP port numbers in the packet header. A zero value is a wildcard and indicates that any port will match; a non-zero value must match the relevant field in the header exactly.

`proto`: criterion for matching the IP protocol number. Transport layer protocols such as TCP and UDP, and other protocols which may be embedded in IP such as ICMP, each have their own protocol numbers (as allocated by the IANA). A zero value is a wildcard; although zero is technically a valid protocol number, it is unused in IPv4 so the packet classifier will never see packets which actually have the protocol number set to zero.

`minlength`, `maxlength`: criteria defining a range of packet lengths which the packet must lie within. Again, zero is a wildcard value for either field. It is acceptable to set just one of these fields to specify a range with no minimum or no maximum.

`stringmatches`: a pointer to a linked list of type `classifier_stringmatch`. This contains any number of arbitrary strings to match, specified by position, length and data. All strings in the list must be satisfied by the packet. A null pointer indicates that there are no such strings; any packet will match.

`target_type`: indicates the type of action that should be performed if a packet matches all of the above criteria. This has the following options:

`ENQUEUE`: append the packet to the queue pointed to by the next field;

`HOOK`: pass the packet to the function pointed to by the next field.

`target`: a union whose actual type depends upon the value of `target_type`.

The actual code for the PTB algorithm, the packet classifier, and related functions and definitions is included in Appendix A.

3.4 Input/output abstraction layer

The I/O abstraction layer deals with any file descriptor (FD) operations required by the other modules. It is specifically tailored to dealing with entire packets, as it is usual to read or write packets atomically when dealing with the Linux networking subsystem.

Arrays of FDs (embedded in structures containing associated metadata) for reading (`io_readfds`) and for writing (`io_writefds`) are maintained by this module. FDs registered for reading have an associated pointer (`callback`) to the function to which packet data read from this FD is to be passed. FDs registered for writing have an associated FIFO queue of packets for transmission; during normal operation this will remain almost empty at all times, but is present to handle the rare situation where the kernel is not in a suitable state to accept a packet. Modules wishing to transmit packets call `io_enqueue` which appends to the relevant FD's queue.

It should be noted that a file descriptor may occur separately in both `io_readfds` and `io_writefds`, in order that it may be used for both reading and writing which are handled independently.

The I/O layer also deals with the correction of the maximum transmission unit (MTU) setting on any network interfaces associated with the FDs it manages — in particular the TUN/TAP interface, although the implementation is not tied to such interfaces and can configure any FD with an MTU. "Message too long" error codes (`EMSGSIZE`) returned by the `write` system call are detected and signalled (using another callback, `mtu_exceeded_cb`) to the module that tried to transmit an over-sized packet. This is likely to be the encapsulation module, `encap.c`; if this tries to prefix more headers to a packet that is already at the maximum size, the packet will become too large for the network to carry atomically. The module responsible should then compute the correct MTU (the Linux networking subsystem provides a path MTU discovery mechanism for this purpose) and send the result to the I/O layer again. Any FDs which have the optional `mtu_update_cb` callback are passed the updated MTU value so that they may reconfigure themselves.

The core of the I/O layer's logic is contained in the `io_select` function, which contains at its core a call to the `select` library function. This is designed to be called either periodically to poll FDs (by setting the timeout parameter to zero) or whenever a wait of a known time is needed; `queue.c` calls `io_select` in both modes. The code for this function and other parts of `io.c` is included in Appendix B.

During the implementation of this module, I discovered what appears to be an inconsistency between the documentation for `select` and its actual

behaviour on my Linux system. The documentation states that the timeout value passed (by pointer) in the `select` call is updated to reflect the amount of time of the timeout remaining, which will be either zero (if no activity on the FDs was detected before the timeout) or between zero and its original value. However I found that frequently the timeout was set to an incorrect value, typically double its original value. As a result I had to implement my own method to discover how much of the timeout was remaining.

3.5 Summary

Overall, the implementation went smoothly and without any major problems. The result was a working system which operated in the manner intended and could be tested as a whole in a realistic scenario. The procedure and results of such testing will be detailed in the next chapter.

CHAPTER 4

Evaluation

During the implementation process, I performed basic unit testing in order to ensure that each module individually behaved as intended. As each module was completed, I checked that it operated correctly alongside all previously implemented code, using skeleton implementations in place of the missing modules.

This chapter describes my approaches to testing the completed system as a whole, and evaluates the results of these tests.

4.1 Test network

In order to test Qtun in a realistic scenario, I have installed it in a scenario as close as possible to its target use case as described in the Introduction and illustrated in Figure 1.1. I set up Qtun nodes on two Linux-based computers, both running version 2.6.15 or newer of the Linux kernel and a recent set of libraries and utilities. The first is a PC which acts as a router for a small home network with a sub-megabit ADSL link to the Internet; it was my attempt to use VoIP on this connection that originally inspired me to propose this project, and all the problems I discussed are easily reproducible on this network. The second node is a colocated server with a 100 Mbit/s Ethernet connection to a high-speed Internet backbone; on this computer I configured Linux packet forwarding to route packets from the Qtun overlay network onto the Internet and vice versa. The round trip time for a small ICMP ping packet sent between these two computers across the Internet and back again is approximately 43 ms when the connection is idle. When the packet size increases to 1280 bytes, the round trip time increases to 80 ms.

In addition to testing Qtun on its own, this setup will allow me to compare Qtun's performance to an existing widely-used alternative, the Linux

Traffic Control system. This system only works at one end of the connection, so is more limited in its capabilities. Although Traffic Control is known to improve latency considerably on this network, I expect Qtun to demonstrate a clear advantage.

4.2 End-user testing

Before beginning this project, I performed some qualitative tests of VoIP on my ADSL connection with the intention of investigating whether Linux Traffic Control could provide an adequate solution. This involved asking users to make calls to a landline from the VoIP phone and hold a brief conversation; this was repeated with the router in various different configurations, and with varying numbers of simultaneous TCP streams on the Internet link.

Calls made on this setup were perceived as similar in quality to a landline-to-landline call when no TCP streams were sharing the connection. With one TCP stream and no QoS, the system became unusable; the delay (which I estimated to be over a second) made conversation impossible and packet loss was sufficient that quality was vastly reduced. During the first few seconds of the call, the line was mostly silent as the majority of packets were being lost before TCP reduced its transmission rate. (It should be remembered that TCP uses packet loss to determine when it should reduce its transmission rate; when the TCP stream is losing packets, the VoIP stream is also, but unlike TCP the VoIP protocol will not retransmit lost packets.)

Linux Traffic Control — configured using an adaption of the Wonder Shaper script [30], which makes use of HTB, SFQ and ingress policing — improved the situation by reducing the delay considerably, but suboptimal audio quality due to dropped packets was still observed. The problems during the first few seconds were also still present to an extent: as Traffic Control does not have direct control over the incoming data, it takes some time for it to cause the competing TCP stream to throttle back. Adding another TCP stream made the problem considerably worse to the extent that the line became unusable again.

I found that the only way to make the first few seconds of the call acceptable using Linux Traffic Control was to permanently rate-limit non-VoIP traffic such that capacity was always reserved for VoIP. In this situation, TCP streams are throttled back to the correct rate before the VoIP stream begins. This is, of course, highly inefficient — especially as the VoIP system requires a sizeable proportion of the total link capacity. Additionally, the problem is not solved entirely, as TCP streams which start during an active VoIP call cause a momentary reduction of quality as packets are lost.

Recent similar tests using Qtun in place of Linux Traffic Control, with Qtun configured to prioritise all UDP traffic over TCP (with a ceiling rate on UDP traffic to avoid starvation), were highly successful — due to the hard bidirectional prioritisation, a VoIP call with simultaneous TCP streams was consistently comparable in quality to one using an idle ADSL link. The VoIP system in this setup was at all times useable for conversation, and had good audio quality. The TCP streams did suffer minor problems due to the sudden changes in available capacity, however, but recovered after a short delay.

4.3 Quantitative testing methods

4.3.1 Flow graphing

I will be using Ethereal [31] to analyse the packets passing across the ADSL connection under test. Ethereal has a flexible graphing subsystem which can plot packet rates or data rates for packets matching user-defined filters. With suitable filters, I will use this to produce graphs showing how the capacity of the link is shared between flows as the number and type of flows varies over time.

4.3.2 Sample traffic generation

Due to the inconvenience of repeatedly making manual VoIP phone calls in order to test Qtun, I have implemented an application — VoIPsim — which simulates a simple VoIP-like stream, by sending UDP packets of a constant size (1280 bytes) at a constant configurable rate in both directions between two nodes. Whilst generating this stream, the simulator gathers statistics about packet loss and latency. This data will be used in combination with traffic graphs to analyse flow behaviour.

Incidentally, VoIPsim makes extensive use of the general-purpose logging module and input/output abstraction layer from Qtun for a purpose which I did not originally foresee, but for which they proved to be highly suitable.

TCP streams to compete with the simulator's UDP traffic will be generated using `wget` to fetch large files via HTTP from a web server running on either node.

When graphing the behaviour of Qtun compared to other systems I will need to generate a fixed pattern of traffic, with various flows starting and stopping at different times. This will be handled by a short shell script In order to generate a fixed pattern of traffic consistently, which schedules the following processes and captures the incoming packets using `tcpdump` [32]:

- TCP download 1: begun immediately, and run for 90 seconds
- TCP download 2: begun 40 seconds from script start, and run for 30 seconds
- VoIPsim UDP stream transmitted at a fixed rate of 20 kBytes/sec: begun 20 seconds from script start, and run for 90 seconds

This allows the behaviour of the UDP stream to be graphed on its own, and with one or two concurrent TCP streams.

4.3.3 Problems encountered during testing

During testing I found three minor problems with the test network. The first was that an upgrade to a faster connection rate had been partially completed; the modem was synchronising its downstream connection to the telephone exchange at around 800 kbit/s, but the ISP was still enforcing a 512 kbit/s transfer rate. This had a beneficial effect on latency: the buffers in the ISP's modem remained mostly empty as it could always transmit data faster than it would arrive, thus solving (entirely coincidentally) one of the problems I set out to solve with Qtun at the expense of available capacity. However, I required graphs illustrating the problems associated with buffering on the line; I hence reversed the direction of the test, as the upstream connection was not affected and exhibited the latency problems I had previously observed on the downstream link. The test flows would now be sent from the home network to the colocated server, rather than vice versa. Since Qtun is symmetric, it will be unaffected, but Linux Traffic Control would have to be configured on the colocated server rather than the home network in order to mimic the behaviour of a setup in which the majority of traffic is incoming (i.e. downloads) as is required.

The second problem is ATM VC congestion (as detected and reported by my ADSL modem) and can I believe be attributed to contention within the network of my telephone service provider. (The ADSL connection to my ISP is based upon an ATM virtual circuit (VC), upon which is layered IP.) Such congestion appears to cause increased packet loss and delay. As the congestion varies with the time of day, this problem was avoided by repeating tests early in the morning when contention is lower.

The final problem occurred when testing Linux Traffic Control configured using Wonder Shaper; I would frequently see that the order of packets in the test UDP stream was changed considerably between transmission and reception, resulting in some packets being delayed by several seconds. This appears to be an artefact of Traffic Control's SFQ implementation (which may have mistakenly classified my single UDP stream as several flows sharing

a link). SFQ is not relevant to my tests as it aims to solve an orthogonal problem — it deals with the scheduling of individual flows within a class (see Section 2.3.2), rather than scheduling the classes atomically — and can therefore be replaced with a FIFO buffer without decreasing the validity of my results. Removing SFQ did have the effect of alleviating the unwanted packet reordering.

4.4 Discussion of results

I used the procedures detailed above to produce graphs of the data rate (in bytes per second) of the automatically-generated flows on my test network, under three different setups: without a QoS system, with Linux Traffic Control ingress policing (configured to attempt to limit incoming traffic to use slightly less than the available capacity), and using Qtun.

4.4.1 No QoS

Figure 4.1 shows that with no QoS system in place, the UDP stream is received at a very uneven rate when competing with a TCP stream for bandwidth, despite being transmitted at a constant rate. The situation becomes considerably worse when a second concurrent TCP stream is added.

When the UDP stream first begins to be received, it takes a significant amount of time (several seconds) to reach a rate close to that at which it is being transmitted. This correlates with the symptoms users noticed on this setup: specifically, that the first few seconds of the call are almost silent. Similar behaviour occurs when the second TCP stream starts.

While TCP streams are present, the received UDP packet rate fluctuates and is frequently considerably below its transmission rate. The measured UDP packet round trip time varied between 80 and 2900 milliseconds, with a mean of 1500. If this were a VoIP stream, it would likely be unusable due to the delay alone.

Relatively few packets were lost entirely — around 2% on average across several runs — however a VoIP system would likely treat significantly-delayed packets as lost.

4.4.2 Linux ingress policing

As shown in Figure 4.2, the Linux Traffic Control ingress policing algorithm improved the behaviour of the system considerably. There is still jitter in the UDP stream, but this is considerably reduced compared to the previous

Key: **red**: TCP stream 1; **green**: TCP stream 2; **blue**: fixed-rate UDP stream; **black**: total traffic

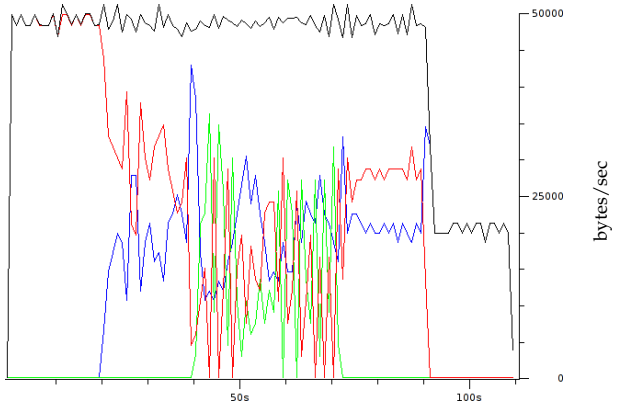

Figure 4.1: Graph of flow rate over time without QoS

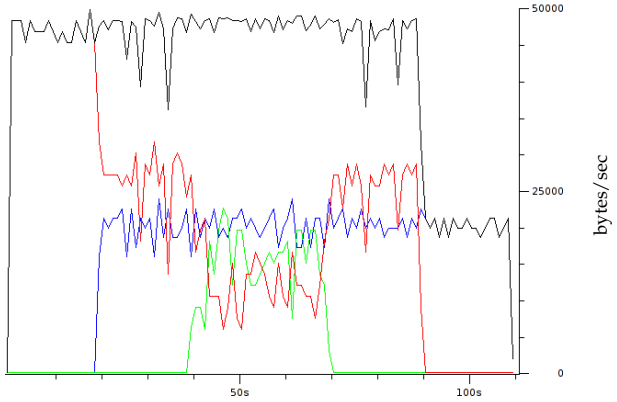

Figure 4.2: Graph of flow rate over time using Linux ingress policing

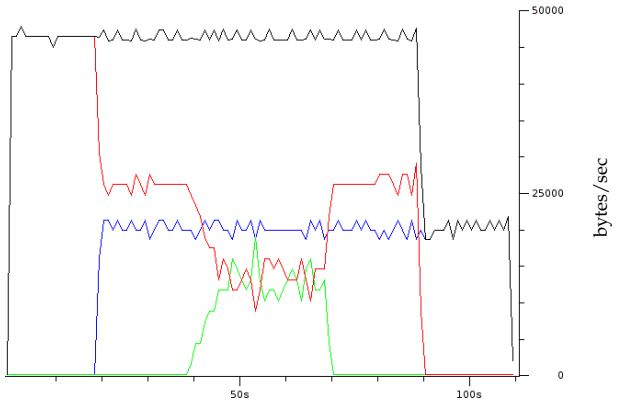

Figure 4.3: Graph of flow rate over time using Qtun

setup as the TCP traffic has been made less aggressive for capacity due to the dropped packets. The measured UDP round trip time varied between 80 and 500 milliseconds, with a mean of 200, which is more likely to be useable for VoIP though is close to the limit at which the latency becomes an irritation.

However, the packets dropped by the ingress policing algorithm will have a noticeably detrimental affect call quality — Linux Traffic Control does not provide a means of dropping only TCP packets in order to police a stream comprising both TCP and UDP, so the dropping of UDP packets is unavoidable. Around 8% of all packets were dropped on average, though this Figure varied over time and would occasionally be much higher for a short time as TCP increased its transmission rate.

It should also be noted that the graph shown is the one exhibiting the best behaviour after running the test several times and with slightly different configurations of the ingress policer. This system is very sensitive to small changes in configuration or conditions, so it is likely that the behaviour of such a system implemented in a realistic setting would at times be considerably worse.

4.4.3 Qtun

It is immediately obvious from Figure 4.3 that all flows have considerably less jitter in the setup using Qtun. The UDP stream in particular has very little jitter. The line on the graph remains slightly jagged due to inaccuracies in Ethereal's graph-plotting system, which both plots at discrete time intervals and assumes that packets arrive instantaneously. When the two TCP streams are both active, they are only competing with each other to use the remaining link capacity after the UDP stream has been accounted for. UDP round-trip time varied between 80 and 200 milliseconds, with a mean of 120. This is only a little over the the minimum round trip time possible on this test network, and is entirely suitable for VoIP use.

The Qtun configuration in this setup is very simple. Two classes are present: a high priority class containing all UDP packets, and a lower priority class for all other traffic.

The disadvantages to the use of Qtun over no QoS are minor: throughput will be marginally lower, as traffic must be throttled to a slightly slower rate than the channel's raw capacity; the lack of large buffers on the link also means that sudden high-rate packet bursts are likely to undergo packet loss rather than smoothing and delay. This latter point is by design, however: low latency is more important for interactive applications than the smoothing of bursts.

CHAPTER 5

Conclusion

I have implemented a general-purpose bidirectional traffic scheduling system, Qtun, capable of effective implementation of a wide range of quality of service policies. In particular, in the scenario in which I aimed to improve performance — the concurrent use of UDP-based voice-over-IP and TCP on the same ADSL link — Qtun is undeniably an improvement on existing QoS systems, as illustrated in the Evaluation chapter. Using Qtun, it is possible to use a low-capacity Internet link for voice-over-IP applications regardless of other activity on the link. This was not possible in a reliable manner using existing unidirectional QoS architectures such as Linux Traffic Control.

The implementation I ended up choosing was not quite the one I originally suggested in my proposal; rather than being based on an existing VPN application, Qtun implements its own IP-over-UDP tunnelling protocol. From the implementation perspective, this gave me more freedom to control exactly what packets are sent; from the user's perspective, Qtun can be used in combination with any VPN application (or none), and hence flexibility of the system is increased.

At the core of Qtun is my Prioritised Token Bucket algorithm, which is responsible for the scheduling of packets. This draws upon concepts and ideas from existing packet scheduling algorithms, but is unique to Qtun. The algorithm is presented in detail in Section 3.3. I believe this presents a very flexible yet easy-to-understand architecture for specifying a QoS policy.

5.1 Future directions

As it stands, Qtun is a working prototype. Its main limitation is the requirement for every user to have administrative access to a server with a fast Internet connection, and the knowledge to configure it by editing program source

files and recompiling. If Qtun is to be used to provide a real-world service to users, a few changes would need to be made.

First and foremost, a Qtun node designated as a server would have to be capable of sustaining connections to several independent other nodes, each of which may have a different desired QoS policy. This would likely require the use of multiple threads to maintain separate state for each client node (in particular the queues should be independent to avoid different nodes interfering with each other). Linux does provide a C threading library, but making use of this in Qtun would have required a large amount of additional boilerplate code; I considered a multithreaded implementation to be beyond the scope of my project.

Remote configurability would also be important. This would be possible by adding further opcodes to the existing RMI scheme (see Section 3.1.4), to add or remove queues or packet classifier entries, or to change global settings (such as token rate). However, a naïve implementation could introduce security concerns: it may become possible for a malicious user to perform a denial-of-service attack on the server or on other nodes by crafting an unusual queueing configuration (for example, one which never transmits a packet and queues indefinitely would eventually exhaust the server's memory). A full security audit would be required before untrusted users could be allowed to use such a remote configuration system.

A potentially useful extra feature which could allow Qtun to be used on links whose capacity is unknown or variable would be automatic recalibration of the correct token rate. This could be done simply by transmitting as fast as possible briefly, and detecting the rate the connection can sustain; however this is wasteful of bandwidth (which may be a major problem if bandwidth is expensive) and it may interrupt flows which are legitimately using the link. Instead, it may be possible to adjust the token rate gradually to match the link's capacity by monitoring the usage patterns on the overlay and the resulting additional latency. Brakmo et al [33] applied similar techniques to TCP to form a new flow control algorithm, TCP Vegas; doing the same for Qtun would likely involve considerable work modelling and analysing protocol behaviour.

5.2 Final words

I believe my project has been a great success, and have both enjoyed it and learnt much from it. A system based on Qtun would have the potential to vastly increase the utility of most home Internet connections, and would open up several new possibilities to users.

Bibliography

[1] J. Postel. RFC 791: Internet Protocol, September 1981. `http://rfc.net/rfc0791.txt`.

[2] F. Andreasen and B. Foster. RFC 3435: Media Gateway Control Protocol (MGCP) version 1.0, January 2003. `http://rfc.net/rfc3435.txt`.

[3] British Telecommunications plc. BT Broadband Voice. `http://www.btbroadbandvoice.com/`.

[4] International Organisation for Standardisation. ISO 7498:1984 Open Systems Interconnection — basic reference model.

[5] P. Almquist. RFC 1349: Type of service in the Internet Protocol suite, July 1992. `http://rfc.net/rfc1349.txt`.

[6] J. Postel. RFC 793: Transmission Control Protocol, September 1981. `http://rfc.net/rfc0793.txt`.

[7] J. Postel. RFC 768: User Datagram Protocol, August 1980. `http://rfc.net/rfc0768.txt`.

[8] E. Klemmer. Subjective evaluation of transmission delay in telephone conversations. *Bell Systems Technical Journal*, 46:1141–1147, 1967.

[9] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[10] J. Nagle. RFC 970: On packet switches with infinite storage, December 1985. `http://rfc.net/rfc0970.txt`.

[11] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89*, pages 1–12, 1989.

[12] P. E. McKenney. Stochastic fairness queueing. In *INFOCOM*, pages 733–740, 1990.

[13] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM '95*, pages 231–242, 1995.

[14] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

[15] A. N. Kuznetsov and B. Hubert. Class-based queueing details. tc-cbq-details(8) manual page. `http://www.lartc.org/manpages/tc-cbq-details.html`.

[16] M. Devera. Hierarchical token bucket theory. `http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm`.

[17] J. L. Valenzuela, A. Monleon, I. San Esteban, M. Portoles, and O. Sallent. A hierarchical token bucket algorithm to enhance QoS in IEEE 802.11: proposal, implementation and evaluation. In *Vehicular Technology Conference, 2004, VTC2004-Fall. 2004 IEEE 60th*, volume 4, pages 2659–2662, 2004.

[18] R. Gilligan and E. Nordmark. RFC 1933: Transition mechanisms for IPv6 hosts and routers, April 1996. `http://rfc.net/rfc1933.txt`.

[19] B. Carpenter and C. Jung. RFC 2529: Transmission of IPv6 over IPv4 domains without explicit tunnels, March 1999. `http://rfc.net/rfc2529.txt`.

[20] J. Cameron, M. Gillham, and P. Howarth et al. Poptop — open source PPTP server. `http://www.poptop.org/`.

[21] J. Yonan. OpenVPN — an open source ssl vpn solution. `http://openvpn.net/`.

[22] G. Sliepen and I. Timmermans. tinc VPN daemon. `http://www.tinc-vpn.org/`.

[23] S. Kent and R. Atkinson. RFC 2401: Security architecture for the Internet Protocol, November 1998. `http://rfc.net/rfc2401.txt`.

[24] M. Krasnyansky. Universal TUN/TAP driver. `http://vtun.sourceforge.net/tun/`.

[25] Free Software Foundation. GNU Compiler Collection. http://gcc.gnu.org/.

[26] The SCons Foundation. SCons: a software construction tool. http://www.scons.org/.

[27] Free Software Foundation. GNU C Library. http://www.gnu.org/software/libc/.

[28] CollabNet. Subversion version control system. http://subversion.tigris.org/.

[29] Edgewall Software. Trac: Integrated scm and project management. http://www.edgewall.com/trac/.

[30] B. Hubert. Wonder Shaper. http://lartc.org/wondershaper/.

[31] G. Combs et al. Ethereal: a network protocol analyser. http://www.ethereal.com/.

[32] V. Jacobson, C. Leres, S. McCanne, and the tcpdump.org team. tcpdump, a network traffic dumper. http://www.tcpdump.org/.

[33] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

APPENDIX A

Code for PTB algorithm

This appendix contains an abridged version of `queue.c`, which contains the code for the main loop of the Prioritised Token Bucket algorithm, the packet classifier, and related functions and definitions.

A.1 queue.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <netinet/in.h>
#include <time.h>
#include <sys/time.h>

#include "qtun.h"

#define QUANTUM 10000 /* usec */
#define TOKENRATE 640 /* bytes per quantum */
#define TOKENMAX 1500 /* maximum number of unused tokens we can retain after a quantum */
#define PERMITMAX 1500 /* maximum number of tokens in a per-queue permit */
#define GUARANTEEMAX 1500 /* maximum number of tokens in a per-queue guarantee */
#define DEFQUEUESIZE 9000 /* default maximum length of a single queue in bytes; >=MTU! */

typedef int token_t;

struct classifier_stringmatch {
·   struct classifier_stringmatch* next;
·   size_t pos;
·   size_t len;
·   char* data;
};
```

```
/* For {src,dst}_port and proto, 0 is a wildcard. */
struct classifier_entry {
·   struct classifier_entry* next;
·   struct classifier_entry* prev;
·   in_addr_t src_ip, src_netmask, dst_ip, dst_netmask;
·   in_port_t src_port, dst_port;
·   uint8_t proto;
·   unsigned int minlength, maxlength;
·   struct classifier_stringmatch* stringmatches;
·   enum { ENQUEUE=0, HOOK } target_type;
·   union {
·   ·   struct queue_node* queue;
·   ·   queue_hook hook;
·   } target;
} *classifier_list_head, *classifier_list_tail;

struct queue_node {
·   struct queue_node* next;
·   struct queue_node* prev;
·   unsigned int guaranteed_rate; /* bps */
·   unsigned int ceiling_rate; /* bps */
·   unsigned int max_queue_size;
·   enum { DROP=0 } excess_mode;
·   struct qtun_listpacket* queue_head; /* Remove here */
·   struct qtun_listpacket* queue_tail; /* Add here */
·   unsigned int queue_size;
·   token_t guarantee; /* temporary storage */
·   token_t permit_tokens; /* temporary storage */
·   _Bool permit_exceeded; /* temporary storage */
·   char name[16];
} *queue_list_head, *queue_list_tail;

struct classifier_entry* add_classifier_entry
·   ( struct classifier_entry* classifier_entry_local, _Bool add_at_head );

struct queue_node* add_queue_node
·   ( struct queue_node* queue_node_local, _Bool add_at_head );

struct queue_node* classify_packet( void* pkt, unsigned int len )
{
·   uint8_t* cpkt = pkt; /* for accessing pkt on byte boundaries */
·   uint32_t* wpkt = pkt; /* for accessing pkt on word boundaries */
·   unsigned int ihl = cpkt[0] & 0x0f; /* IP header length in words, min. 5 */
·   for ( struct classifier_entry* iter = classifier_list_head; iter; iter=iter->next ) {
·   ·   if ( iter->src_netmask && ( ntohl( wpkt[3] ) & iter->src_netmask ) != iter->src_ip ) continue;
·   ·   if ( iter->dst_netmask && ( ntohl( wpkt[4] ) & iter->dst_netmask ) != iter->dst_ip ) continue;
·   ·   if ( iter->src_port && ( ntohl( wpkt[ihl] ) >> 16 ) != iter->src_port ) continue;
·   ·   if ( iter->dst_port && ( ntohl( wpkt[ihl] ) & 0x0000ffff ) != iter->dst_port ) continue;
·   ·   if ( iter->proto && cpkt[9] != iter->proto ) continue;
·   ·   if ( 0 < iter->minlength && len < iter->minlength ) continue;
·   ·   if ( 0 < iter->maxlength && len > iter->maxlength ) continue;
·   ·   if ( iter->stringmatches ) {
·   ·   ·   _Bool fail = false;
·   ·   ·   for ( struct classifier_stringmatch* sm = iter->stringmatches; sm; sm=sm->next )
·   ·   ·   ·   if ( 0!=memcmp( ( ( uint8_t* ) pkt ) + sm->pos, sm->data, sm->len ) ) {
·   ·   ·   ·   ·   fail = true;
```

```
· · · · · · break;
· · · · · }
· · · · if ( fail ) continue;
· · · }
· · · /* If we reached here, everything matches. */
· · · /* If target is a hook, run it and continue, otherwise return the target queue. */
· · · if ( iter->target_type == HOOK ) {
· · · · debug( "calling classifier hook 0x%x", iter->target.hook );
· · · · ( *iter->target.hook ) ( pkt, len );
· · · } else
· · · · return iter->target.queue;
· · }
· · /* Shouldn't reach here. Try to cope anyway. */
· · warning( "packet matched no classifiers!" );
· · return queue_list_tail;
}

void enqueue_packet( void* pkt, unsigned int len )
{
· · struct queue_node* target_queue = classify_packet( pkt, len );
· · if ( target_queue->queue_size + len >= target_queue->max_queue_size ) {
· · · · /* Queue full */
· · · · switch ( target_queue->excess_mode ) {
· · · · · · case DROP:
· · · · · · · · debug( "queue is full, dropping packet" );
· · · · · · · · break;
· · · · · · default:
· · · · · · · · error( "Unimplemented excess mode %d", target_queue->excess_mode );
· · · · }
· · · · return;
· · }

· · struct qtun_listpacket* lpkt = malloc( sizeof( struct qtun_listpacket ) );
· · memcpy( lpkt->data, pkt, len );
· · lpkt->length = len;
· · lpkt->next = NULL;

· · if ( 0 == target_queue->queue_size ) {
· · · · lpkt->prev = NULL;
· · · · target_queue->queue_head = lpkt;
· · · · target_queue->queue_tail = lpkt;
· · · · target_queue->queue_size = len;
· · } else {
· · · · lpkt->prev = target_queue->queue_tail;
· · · · target_queue->queue_tail->next = lpkt;
· · · · target_queue->queue_tail = lpkt;
· · · · target_queue->queue_size += len;
· · }
}

void queue_setup()
{
· · /* Example queueing configuration */

· · /* Add a default queue */
· · struct queue_node *queue_node_default = add_queue_node(
```

```
        &( struct queue_node ) {
            .name = "Default"
        }, false );
    add_classifier_entry(
        &( struct classifier_entry ) {
            .target.queue = queue_node_default,
        }, false );

    /* Prioritise UDP */
    struct queue_node *queue_node_udp = add_queue_node(
        &( struct queue_node ) {
            .name = "UDP"
        }, true );
    add_classifier_entry(
        &( struct classifier_entry ) {
            .target.queue = queue_node_udp,
            .proto = IPPROTO_UDP,
        }, true );

    /* Prioritise ICMP above all else */
    struct queue_node *queue_node_icmp = add_queue_node(
        &( struct queue_node ) {
            .name = "ICMP"
        }, true );
    add_classifier_entry(
        &( struct classifier_entry ) {
            .target.queue = queue_node_icmp,
            .proto = IPPROTO_ICMP,
        }, true );
}

void queue_main_loop()
{
    struct queue_node* cndq;
    token_t sparetokens = 0;
    token_t guarantee;
    int waittime;
    _Bool morework;
    struct timeval tv, endtime;
    _Bool first;

    while ( 1 ) {
        gettimeofday( &endtime, NULL );
        if ( endtime.tv_usec + QUANTUM > 1000000 ) {
            endtime.tv_sec++;
            endtime.tv_usec += QUANTUM - 1000000;
        } else {
            endtime.tv_usec += QUANTUM;
        }

        sparetokens += TOKENRATE;

        /* preallocate tokens where they've been guaranteed; compute permits */
        for ( cndq = queue_list_head; NULL != cndq; cndq = cndq->next ) {
            if ( cndq->guaranteed_rate > 0 ) {
                guarantee = cndq->guaranteed_rate * QUANTUM / 1000000;
```

```
        cndq->guarantee += guarantee;
        if ( cndq->guarantee > GUARANTEEMAX )
            cndq->guarantee = GUARANTEEMAX;
        sparetokens -= guarantee;
    } else {
        cndq->guarantee = 0;
    }
    if ( cndq->ceiling_rate > 0 ) {
        cndq->permit_tokens += cndq->ceiling_rate * QUANTUM / 1000000;
        if ( cndq->permit_tokens > PERMITMAX )
            cndq->permit_tokens = PERMITMAX;
    } else {
        cndq->permit_tokens = 0;
    }
    cndq->permit_exceeded = false;
}

do {
    morework = false;
    /* pick queue to send from */
    for ( cndq = queue_list_head;
          NULL != cndq && ( 0 == cndq->queue_size || cndq->permit_exceeded );
          cndq = cndq->next );
    if ( NULL != cndq ) {
        if ( ( sparetokens + cndq->guarantee ) >= cndq->queue_head->length ) {
            morework = true;
            if ( cndq->ceiling_rate > 0 &&
                 ( cndq->permit_tokens < cndq->queue_head->length ) ) {
                cndq->permit_exceeded = true;
            } else {
                /* Update token counts and dequeue packet */
                if ( cndq->guarantee > 0 ) {
                    cndq->guarantee -= cndq->queue_head->length;
                    if ( cndq->guarantee < 0 ) {
                        sparetokens += cndq->guarantee;
                        cndq->guarantee = 0;
                    }
                } else {
                    sparetokens -= cndq->queue_head->length;
                }
                if ( cndq->ceiling_rate > 0 )
                    cndq->permit_tokens -= cndq->queue_head->length;
                encap_enqueue( cndq->queue_head->data, cndq->queue_head->length );
                cndq->queue_size -= cndq->queue_head->length;
                cndq->queue_head = cndq->queue_head->next;
            }
        }
        io_select( 0 );
    }
} while ( morework ); /* keep repeating until we're not sending more packets */

if ( sparetokens > TOKENMAX )
    sparetokens = TOKENMAX;
assert( sparetokens>=0 );

rrmi_check();
```

```
· · first = true;
· · do {
· · · gettimeofday( &tv, NULL );
· · · waittime = timerop_usec( &endtime, &tv, − );
· · · if ( waittime < 0 ) {
· · · · if ( first ) {
· · · · · warning( "system is TOO SLOW for this quantum! main loop takes %d usec",
· · · · · · · QUANTUM−waittime );
· · · · · io_select( 0 );
· · · · } else
· · · · · break;
· · · } else {
· · · · io_select( waittime );
· · · }
· · · first = false;
· · } while ( 0 < waittime );
· }
}
```

APPENDIX B

Code for I/O abstraction layer

This appendix contains an abridged version of `io.c`, containing the code for the core functions of the input/output abstraction layer, and relevant definitions from its associated header file `io.h`.

B.1 io.h

```
#define MAX_PACKET_SIZE 1600

struct qtun_listpacket {
·   char data[MAX_PACKET_SIZE];
·   unsigned int length;
·   struct qtun_listpacket* prev;
·   struct qtun_listpacket* next;
};

typedef void ( *qtun_io_pktpass_cb ) ( void* /*packet*/, unsigned int /*length*/ );
typedef int ( *qtun_io_writefd_cb ) ( struct qtun_io_writefd* );
typedef void ( *qtun_io_mtu_update_cb ) ( unsigned int /*newMTU*/ );

struct qtun_io_readfd {
·   int fd;
·   qtun_io_pktpass_cb callback;
·   qtun_io_mtu_update_cb mtu_update_cb;
};

struct qtun_io_writefd {
·   int fd;
·   qtun_io_writefd_cb mtu_exceeded_cb;
·   struct qtun_listpacket* queue_head; /* Remove here */
·   struct qtun_listpacket* queue_tail; /* Add here */
·   unsigned int queue_length;
};
```

B.2 io.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "io.h"
#include "log.h"

#define MAX_READ_FDS 32
#define MAX_WRITE_FDS 8

static struct qtun_io_readfd* io_readfds[MAX_READ_FDS];
static int io_readfds_len = 0;

static struct qtun_io_writefd* io_writefds[MAX_WRITE_FDS];
static int io_writefds_len = 0;

static int io_maxfd = 0;

void io_add_readfd( struct qtun_io_readfd* rfd );
void io_add_writefd( struct qtun_io_writefd* wfd );
void io_enqueue( struct qtun_io_writefd* wfd, void* data, unsigned int len );

void io_select( int timeout_usec )
{
·   fd_set rset, wset;
·   int i, r, drop;
·   char buf[MAX_PACKET_SIZE];
·   struct timeval tv;

·   memset( &tv, 0, sizeof( tv ) );

·   FD_ZERO( &rset );
·   if ( io_readfds_len>0 )
·   ·   for ( i=0; i<io_readfds_len; i++ )
·   ·   ·   FD_SET( io_readfds[i]->fd, &rset );

·   FD_ZERO( &wset );
·   if ( io_writefds_len>0 )
·   ·   for ( i=0; i<io_writefds_len; i++ )
·   ·   ·   if ( io_writefds[i]->queue_length>0 ) {
·   ·   ·   ·   if ( io_writefds[i]->queue_length>2 )
·   ·   ·   ·   ·   warning( "backlog of %d packets on writefd %d",
·   ·   ·   ·   ·   ·   ·   io_writefds[i]->queue_length-1, io_writefds[i]->fd );
·   ·   ·   ·   FD_SET( io_writefds[i]->fd, &wset );
·   ·   ·   }

·   if ( timeout_usec<0 ) {
·   ·   if ( 0>select( io_maxfd+1, &rset, &wset, NULL, NULL ) ) {
·   ·   ·   if ( EINTR==errno )
·   ·   ·   ·   return;
·   ·   ·   else
```

```
        fatal( "select() failed: %s", errstr );
    }
  } else {
    tv.tv_sec = 0;
    tv.tv_usec = timeout_usec;
    if ( 0>select( io_maxfd+1, &rset, &wset, NULL, &tv ) ) {
      if ( EINTR==errno )
        return;
      else
        fatal( "select() failed: %s", errstr );
    }
  }

  if ( io_readfds_len>0 )
    for ( i=0; i<io_readfds_len; i++ )
      if ( FD_ISSET( io_readfds[i]->fd, &rset ) ) {
        r = read( io_readfds[i]->fd, &buf, MAX_PACKET_SIZE );
        if ( r == -1 )
          debug( "readfd %d: read failed: %s", io_readfds[i]->fd, errstr );
        else
          io_readfds[i]->callback( buf, r );
      }

  if ( io_writefds_len>0 )
    for ( i=0; i<io_writefds_len; i++ )
      if ( FD_ISSET( io_writefds[i]->fd, &wset ) ) {
        assert( io_writefds[i]->queue_length != 0 );
        assert( io_writefds[i]->queue_head != NULL );
        assert( (io_writefds[i]->queue_length<=1) || (io_writefds[i]->queue_head->next!=NULL) );
        r = write( io_writefds[i]->fd, io_writefds[i]->queue_head->data,
          io_writefds[i]->queue_head->length );
        if ( r == -1 ) {
          if ( errno == EMSGSIZE && io_writefds[i]->mtu_exceeded_cb )
            drop = io_writefds[i]->mtu_exceeded_cb( io_writefds[i] );
          else {
            warning( "writefd %d: write failed (will retry): %s",
              io_writefds[i]->fd, errstr );
            drop = 0;
          }
        } else if ( r != io_writefds[i]->queue_head->length ) {
          warning( "writefd %d: incomplete write (sent %d, length %d); will retry",
            io_writefds[i]->fd, r, io_writefds[i]->queue_head->length );
          drop = 0;
        } else {
          drop = 1;
        }
        if ( drop ) {
          if ( io_writefds[i]->queue_head->next ) {
            io_writefds[i]->queue_head = io_writefds[i]->queue_head->next;
            free( io_writefds[i]->queue_head->prev );
            io_writefds[i]->queue_head->prev = NULL;
          } else {
            assert( io_writefds[i]->queue_length == 1 );
            free( io_writefds[i]->queue_head ); // (head==tail)
            io_writefds[i]->queue_head = NULL;
            io_writefds[i]->queue_tail = NULL;
```

```
· · · · · }
· · · · · io_writefds[i]->queue_length--;
· · · · · assert( (io_writefds[i]->queue_length==0) == (io_writefds[i]->queue_head==NULL) );
· · · · }
· · · }
}

void io_update_mtu( unsigned int newmtu )
{
·  for ( int i=0; i<io_readfds_len; i++ )
·  ·  if ( io_readfds[i]->mtu_update_cb )
·  ·  ·  io_readfds[i]->mtu_update_cb( newmtu );
}
```

APPENDIX C

Project Proposal

C.1 Introduction

My idea is to develop a means of prioritising certain types of incoming and outgoing IP traffic on a computer with a relatively slow Internet link (such as ADSL). The prioritisation would be done based on various critera of the IP packets, such as the protocol (e.g. TCP, UDP, ICMP) and port number, to identify different applications (for example, TCP port 22 is typically used for interactive SSH traffic, which could be given a higher priority than bulk transfers in order to minimise latency on interactive sessions). Additionally, other factors may be taken into consideration: for example, interactive traffic typically has smaller packet sizes compared to bulk transfer traffic, so smaller packets could be prioritised over larger ones.

There are a number of existing QoS systems designed to be run on a single router or end system (see *Starting point*) which maintain a queue of packets according to some queueing discipline. However this approach is only effective for controlling transmitted data; received data (which often exceeds transmitted data in volume, especially when bulk downloads are in progress) will already have passed through the slow link by the time it reaches the system doing QoS (assuming the Internet Service Provider is not running a similar QoS system on its routers, which is generally the case). In the case of TCP streams, dropping occasional packets can cause the transmission control algorithm to throttle the packet rate back, but dropping UDP packets (or other connectionless protocols) is generally not a good idea: once a packet is dropped, it is generally lost forever (unless a higher-layer protocol is capable of retransmission).

A specific aim of this project is to allow UDP-based Voice-over-IP (VoIP) traffic to share an ADSL link effectively with TCP downloads, without inten-

tionally dropping any of the VoIP packets and without allowing the (typically large) queues present in the ADSL modem from introducing unacceptable delay into the audio stream (which will usually occur when the link is oversaturated). The VoIP data typically uses 100-120 kbps in both directions, i.e. around 20% of the download capacity of a 512 kbps ADSL link, so it is inefficient to always reserve part of the link for VoIP traffic when the VoIP system is only used occasionally, especially if more than one VoIP system can share the link.

C.2 Work planned

I intend to solve the problems mentioned above by applying a QoS system at both ends of an overlay network consisting of an IP tunnel to a trusted server with a considerably faster link to the Internet. All traffic passing over the slow link should go through this tunnel; the QoS overlay would therefore contain and control all data flowing through the slow link. The prioritisation is intended to be configurable, and the two ends of the tunnel will communicate metadata between them such as configuration changes or details of streams which must be given higher or lower priority.

The system will be implemented as an add-on or patch to an existing VPN system (see *Starting point*) in order to take advantage of an existing IP tunnelling protocol, and to provide authentication and privacy of tunnelled data as a useful side-effect. However I am currently undecided as to whether all the packet scheduling code should be within the VPN application, or whether this should be offloaded to another program — this decision will be made early in the development process, following some research into the workings of the VPN system I decide to use.

It should be noted that unlike many other QoS systems, such as those proposed by the IETF's Diffserv[1] and Intserv[2] working groups, the system I propose does not need to be supported on any systems except the endpoints of the tunnel; the intermediate routers will see a single flow between the endpoints and do not need to perform any intelligent queueing or scheduling themselves.

[1] `http://www.ietf.org/html.charters/OLD/diffserv-charter.html`
[2] `http://www.ietf.org/html.charters/OLD/intserv-charter.html`

C.3 Resources

I will need a computer with a high-bandwidth Internet link on which to run the server, and a computer with a lower-bandwidth link to act as a client. For the former, I intend to use a computer running Debian Linux in my college room with a 100 Mbps connection to the Trinity Hall network and the University network (CUDN), and hence to the Internet. I have a number of other computers in my room which I could use should one fail. The college permits me to run server software on my computers, so I do not foresee a problem with doing this. However, if this turns out to be inadequate for whatever reason or if I decide to try the system under different conditions, I own a colocated server (also with a 100 Mbps link to the Internet) which could also be used.

For the client computer, I intend to use a computer at my parents' house, which has a 512 kbps PlusNet ADSL link. This computer runs Gentoo Linux and I have SSH access to it. Should this fail, I know a number of other people with ADSL links (generally 2 Mbit) who could probably be persuaded to let me run this system on their computers. I may also attempt to simulate links of various speeds by using other software to limit transmission rate at both ends of the tunnel.

If I have sufficient time, I intend to try the system out on systems slower than ADSL, such as a GPRS mobile phone; for this I would require access to the Computer Lab's GPRS equipment.

C.4 Starting point

C.4.1 Existing VPN systems

There are a number of freely-available, open source VPN systems which I could use as a starting point; the three I have experience of are Poptop PPTPD[3], OpenVPN[4], and TINC[5]. I have been involved in a project to write a logging system for Poptop, so should I choose to use that I would start with basic knowledge of the plugin architecture and an example plugin to work with (however a logging plugin is likely to be very different to one for QoS!).

[3] `http://www.poptop.org/`
[4] `http://openvpn.net/`
[5] `http://www.tinc-vpn.org/`

C.4.2 Existing QoS systems

I have in the past used the QoS system in the Linux kernel (`tc`) in previous experiments to try to prioritise my VoIP phone on an ADSL link. This is a very flexible queueing discipline system which goes beyond what I intend to implement, but is by nature independent of other nodes and has no support for collaborating with another host running `tc` at the far end of a tunnel. I do not intend to copy code directly from `tc` into my project; however I will look at the code and algorithms used for inspiration. A popular script based on this system, "Wonder Shaper"[6], is moderately effective at prioritising interactive traffic over bulk transfers, but assumes that the volume of the traffic being prioritised is low — this is not the case for VoIP traffic, and I have found that Wonder Shaper is not effective at prioritising this.

The Click Modular Router Project[7] also contains basic queueing and packet scheduling modules, which may be of use, but again this is designed for an independent router.

C.4.3 Existing packet classifiers

It is possible that identification of "trigger packets" (see *Detailed description*) will require detailed dissection of a protocol, for example identifying a VoIP system's "start of call" packet. There are systems already in existance which are capable of dissecting many different protocols, such as tcpdump/libpcap[8] and Ethereal[9]; if I need to do protocol dissection in my system I may use code from such an existing system as an example. Alternatively it may turn out that I can identify the packets I need by just looking at one or two bytes at a fixed location in the packet, which should not pose any problem.

I will refer to the relevant standards documents to determine which packets should be trigger packets; for example, my VoIP phone uses MGCP, which is defined in detail in RFC 2705[10].

C.4.4 Programming languages

All of the VPN systems I mentioned above are written in C, so it is likely that much of my code will also have to be written in C if I am to interface with or extend these. I have little experience of writing large programs in C, so I

[6] `http://lartc.org/wondershaper/`
[7] `http://pdos.csail.mit.edu/click/`
[8] `http://www.tcpdump.org/`
[9] `http://www.ethereal.com/`
[10] `http://www.faqs.org/rfcs/rfc2705.html`

intend to learn this as I proceed with this project. The Click Modular Routing Project is written in C++, which I may also have to learn.

C.5 Detailed description

The project will involve implementing a packet classifier to identify packets to be sent with different priorities. This will inspect packet headers and categorise packets according to any number of protocol, port number, size, source, and destination. The packets will then be added to a queue; it is likely that there will be multiple queues for different classes of packet. Packets will be dequeued according to a queueing discipline, at a maximum bit rate to ensure that the slow link is never over-saturated; this avoids queues over which we have no control from filling up, such as queues in an ADSL modem or router.

Additionally, some packets may cause the state of the queueing discipline to change; I will refer to these as "trigger packets". For example, a trigger packet may be used to identify the start of a VoIP call; when this is detected, a fixed amount of capacity could be reserved for the call, ensuring that the call is given absolute priority over all other traffic. TCP connections would be throttled to use the remainder of the link, using a technique such as Random Early Detection (RED)[11].

C.5.1 Examples of queueing disciplines

Below are described a number of queueing disciplines which I will consider for inclusion in my QoS system. A number of these summaries are adapted from the Linux Advanced Routing and Traffic Control (LARTC) manual pages[12].

- **FIFO** (First In, First Out) — very simple, so this will be useful for testing; however this will have no QoS effect as the VPN systems I will be adapting are already FIFO by design. However FIFO can be adapted to do simple QoS by maintaining a number of FIFO queues for different priorities of packet which are emptied in a set order.
- **RED** (Random Early Detection), as mentioned above — this simulates physical congestion by randomly dropping packets when nearing a configured maximum bandwidth allocation, with a probability which

[11] Floyd, S., and Jacobson, V., Random Early Detection gateways for Congestion Avoidance; `http://www.aciri.org/floyd/papers/early.pdf`

[12] `http://www.lartc.org/manpages/`

increases with queue length to ensure that the queue remains relatively short.

- **SFQ** (Stochastic Fairness Queueing) — reorders queued traffic such that each flow (identified by a hash of source address, destination address, and source port) is permitted to send a packet in turn.

- **TBF** (Token Bucket Filter) — ensures that packets are not sent out above a precise maximum bandwidth, by providing the queue with "tokens" (roughly corresponding to bytes) at a fixed rate, and each packet sent consumes a number of tokens; when there are no tokens, no packets are sent until a token arrives.

These all act on one queue, and can be combined in various ways to be effective with multiple queues; for example LARTC's HTB (Hierarchy Token Bucket) is an adaption of TBF which guarantees that higher-priority queues will be allocated tokens first if they need them, with "spare" tokens allocated to lower-priority queues.

C.6 Assessment criteria

C.6.1 Core goals

The primary goal of this project is to create a QoS system which works effectively with simultaneous TCP and non-retransmitting (e.g. UDP, ICMP) traffic. Specifically, it should be capable of throttling a TCP-based download in favour of a UDP stream on an ADSL link, for example VoIP, with minimal increase in end-to-end delay and without causing significant packet loss on the UDP stream. This will be tested both subjectively (by testing whether the quality and delay on the VoIP service is "acceptable", or at least better than in a setup with the QoS system turned off) and objectively (by measuring delay and by gathering packet loss statistics).

It should also be capable of throttling back a TCP download in favour of interactive TCP traffic, such as SSH. This will be tested in a similar way, by subjectively comparing how useable the SSH session is during a download with and without the QoS system, and objectively by measuring latency.

C.6.2 Intended extensions

Depending on time availability, I intend to implement a number of the following extensions:

- Manual configurability of prioritisation, based on packet protocol / source address / destination address / port number etc.
- A limited amount of automatic configuration, for example automatic testing of the capacity of the Internet link (either one-off or repeatedly during operation to deal with variable-capacity channels, such as a shared ADSL link where some traffic will bypass the QoS system)
- Automatic gathering of statistics (such as the effect of traffic volume upon latency) — this may be necessary for objective testing of the system
- Dealing with slower links such as GPRS, which have much lower capacity, non-negligible packet loss, and variable latency
- Extension of the system to allow more than two nodes: for example, a single client connecting to multiple servers and selecting which to use for a particular packet based on proximity to the packet's destination, or multiple clients connecting to a single server.
- If the system is sufficiently complete, I will consider submitting my work to the maintainers of the VPN system I adapted.

C.7 Plan of work

Weeks	Work
Michaelmas term	*Week 1 begins on 22nd October 2005*
1–2	Investigating existing VPN systems, and selecting one to use
3	Gain familiarity with VPN system's code; determine whether project should be implemented entirely within the VPN system or by passing packets through a separate program
4–6	Implement packet classifier
6	**Milestone:** Modified VPN system is capable of determining protocol, size, source, destination, and port of packets
Christmas vacation	
7	Catch-up buffer
8	Implement packet queueing and dequeueing (initially FIFO)
9-10	Begin implementing queueing discipline
Lent term	*Week 11 starts on 14th January 2005*
11	**Milestone:** The modified VPN system will still forward packets correctly after passing them through a FIFO queue
11–14	Complete implementation of queueing discipline
12	**Deliverable:** Progress Report
13	**Milestone:** Core goals complete
15–16	Testing & collecting performance statistics
17–18	Writing dissertation
18	**Deliverable:** Draft dissertation (without Evaluation chapter)
Easter vacation	*(primarily set aside for revision)*
19	Completing draft dissertation
Easter term	*Week 20 starts on 22nd April 2005*
20	**Deliverable:** Draft dissertation
20–21	Finalising dissertation
23	**Deliverable:** Final dissertation